Concurrent SSA Form in the Presence of Mutual Exclusion [*]

Diego Novillo Ron Unrau Jonathan Schaeffer
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada
E-mail: {diego,unrau,jonathan}@cs.ualberta.ca

Abstract

Most current compiler analysis techniques are unable to cope with the semantics introduced by explicit parallel and synchronization constructs in parallel programs. In this paper we propose new analysis and optimization techniques for compiling explicitly parallel programs that use mutual exclusion synchronization. We introduce the CSSAME form, an extension of the Concurrent Static Single Assignment (CSSA) form that incorporates mutual exclusion into a data flow framework for explicitly parallel programs. We show how this analysis can improve the effectiveness of constant propagation in a parallel program. We also modify a dead-code elimination algorithm to work on explicitly parallel programs. Finally, we introduce lock independent code motion, a new optimization technique that attempts to minimize the size of critical sections in the program.

1. Introduction

Although recent advances in parallelizing compilers and data-parallel languages have been impressive [4, 8], there are important problem domains for which parallelizing the best sequential algorithm or data layout yields sub-optimal performance relative to an implementation that is explicitly parallel from the outset. Furthermore, popular systems like Java incorporate parallel constructs at the language level and commodity multiprocessors are becoming increasingly popular. For these reasons, we believe that there is a need for compilers that accept explicitly parallel programs, and that the demand for such compilers will increase.

To correctly compile and optimize explicitly parallel programs the compiler must have an innate knowledge of the parallelism in the program and the semantics of synchronization primitives. In addition to the standard optimization techniques used by sequential compilers, an optimizing parallel compiler should exploit the parallel structure of the program to achieve better performance. Unfortunately, standard optimization techniques used in sequential programs cannot be directly applied to explicitly parallel programs because they may generate incorrect transformations [11]. This has motivated recent developments that have started to uncover the potential benefits of analysis and optimization techniques for explicitly parallel programs [3, 7, 13]. Like any incipient technology, these techniques are still in their primitive stages, especially when compared to their sequential counterparts.

Initial work by Shasha and Snir proposed re-ordering memory references in a program to increase concurrency while maintaining the sequential consistency dictated by the code [13]. Midkiff and Padua demonstrated that a direct application of optimization techniques designed for sequential languages fail on explicitly parallel programs [11]. Grunwald and Srinivasan developed data-flow equations to compute reaching definition information on explicitly parallel programs with `cobegin/coend` parallel sections [3]. However, their work only deals with a weak memory consistency model dictated by the PCF Fortran standard. Parallel sections are required to be data independent; memory updates are done at specific points in the program using the `copy-in/copy-out` model. Synchronization is limited to event-based synchronization using `Set` and `Wait` operations. Knoop, Steffen and Vollmer developed a bitvector analysis framework for parallel programs with shared memory and interleaving semantics [6]. They show how to adapt standard optimization algorithms to their framework. However, they do not incorporate synchronization operations in their analysis. Lee, Midkiff and Padua propose a Concurrent SSA framework (CSSA) for explicitly parallel programs and interleaving memory semantics [7]. They only consider event-based synchronization and impose some restrictions on the input program.

A major limitation of existing techniques for optimizing explicitly parallel programs is the restricted knowledge about synchronization in the program. To the best of our knowledge, the only synchronization construct recognized is a subset of event-based synchronization (i.e., `Set` and

[*] To appear in the 1998 International Conference on Parallel Processing (ICPP'98), Minneapolis, Minnesota, August 1998.

Wait usually with no Clear). We see this as a severe limitation because event synchronization can only be used to describe a small class of parallel algorithms. One of the goals of our work is to incorporate knowledge about common synchronization structures into the compiler so it can perform more aggressive optimizations. As a first step to that goal, we have extended the Concurrent Static Single Assignment (CSSA) form [7] to handle mutual exclusion synchronization. Specifically, we

- extend the concurrent control flow graph used by Lee *et al.* (Section 3.1) and show how to detect mutual exclusion synchronization in a parallel program (Section 3.3),
- introduce the CSSAME[1] form, an extension to the CSSA form to account for the semantics introduced by mutual exclusion synchronization (Section 4),
- show how CSSAME can improve the effectiveness of the Concurrent Sparse Conditional Constant (CSCC) propagation algorithm [7] (Section 5.1),
- adapt a sequential dead-code elimination algorithm to work on explicitly parallel programs (Section 5.2), and
- introduce *lock independent code motion*, a new optimization technique for explicitly parallel programs which attempts to reduce the size of mutual exclusion sections in the program (Section 5.3).

2. Our approach

In an explicitly parallel program with interleaving memory semantics, the use of a shared variable v can be reached by any definition of v in another concurrent thread. However, mutual exclusion may prevent some variable definitions from being visible in other threads. For example, consider the code fragment in Figure 1. If we ignore the mutual exclusion regions created by the locks we will conclude that the definition for variable a in thread T_0 can reach both uses of a in thread T_1. However, the synchronization used in the program serializes the references to a so that the assignment to a in T_0 cannot reach the second use of a in T_1. Therefore, the call to function $g()$ in T_1 will always be executed with $a = 3$.

Understanding mutual exclusion has important implications from an optimization point of view because it allows the compiler to reduce the number of data dependencies that need to be considered. It also allows the compiler to conservatively validate the synchronization structures expressed inside the code. This paper focuses on the former; future work will also investigate correctness and user interface issues.

To determine the effects of mutual exclusion on the dataflow of the program the compiler must recognize which sections of the program execute under the protection of a lock. We base our analysis on the concept of *mutex structures* first introduced by Masticola and Ryder in their work on non-concurrency analysis [10]. Basically, a mutex structure is associated with each lock variable used in the program and it contains the sets of flow graph nodes that are guaranteed to execute under the protection of the associated lock variable. Once mutual exclusion information is gathered into mutex structures, we modify the Concurrent SSA (CSSA) form proposed by Lee *et al.* [7] to account for it.

```
cobegin /* Begin concurrent execution */
  T 0: begin              /* Launch thread T0 */
    Lock(L);
    a = a + b;
    Unlock(L);
  end

  T 1: begin              /* Launch thread T1 */
    f(a);
    Lock(L);
    a = 3;                /* This kills the assignment to a in T0 */
    b = b + g(a);         /* Variable a is always 3 */
    Unlock(L);
  end
coend
```

Figure 1. Mutual exclusion can reduce data dependencies across threads in a parallel program.

Explicitly parallel programs start as a single thread of computation. New threads are logically created when execution reaches a parallel section. Although the creation, placement and scheduling of threads is not significant for our research, the compiler must be able to recognize parallel sections in the code. We assume that threads run in a shared address space with interleaving semantics (i.e., updates to shared memory made by one thread are immediately visible to other threads). There are a variety of mechanisms for expressing parallel activity. Some examples include cobegin/coend constructs, explicit fork statements, parallel loops, etc. In this paper parallel sections are specified using cobegin/coend constructs (Figure 1).

Mutual exclusion is used to serialize references to shared variables in the program. We will assume, without loss of generality, that programmers use standard Lock and Unlock instructions to serialize access to shared variables.

3. Mutual exclusion analysis

3.1. Parallel Flow Graphs

We introduce the Parallel Flow Graph (PFG), an extension to the Concurrent Control Flow Graph (CCFG) [7] that also represents mutual exclusion synchronization. In addition to the directed synchronization edges in the original CCFG, we incorporate undirected mutex synchronization edges which represent mutual exclusion constraints and do not enforce a specific execution order. Each Lock and Unlock operation is represented by a separate node in the PFG. Mutex synchronization edges join Lock and

[1] Pronounced *sesame*.

```
a = 0;
b = 0;
cobegin
  T 0: begin
    Lock(L);
    a = 5;
    b = a + 3;
    if (b > 4) {
      a = a + b;
    }
    x = a;
    Unlock(L);
  end

  T 1: begin
    Lock(L);
    a = b + 6;
    y = a;
    Unlock(L);
  end
coend
print(x);
print(y);
```

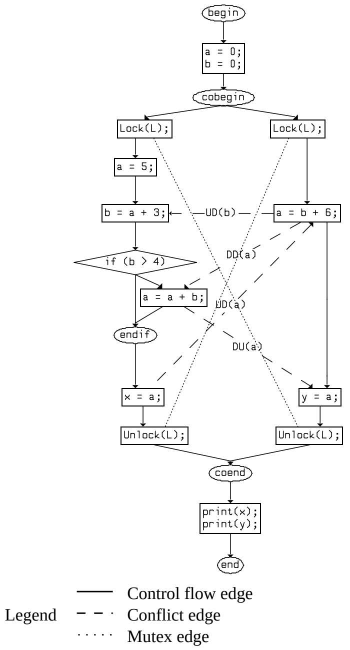

Figure 2. A program and its PFG.

Unlock nodes that operate on the same variable in concurrent threads.

Definition 1 A *Parallel Flow Graph (PFG)* is a directed graph $G = \langle N, E, Entry_G, Exit_G \rangle$ such that:

1. N is the set of parallel basic blocks in the program.
2. $Entry_G$ and $Exit_G$ are the unique entry and exit points of the program.
3. Lock and Unlock operations are represented by their own nodes.
4. $E = E_{ct} \bigcup E_{sy} \bigcup E_{cf}$ is the set of edges in the graph such that E_{ct} is the set of control flow edges. These edges have the same meaning as in a sequential control flow graph (CFG). $E_{sy} = E_{mutex} \bigcup E_{dsync}$ is the set of synchronization edges. Two different kinds of synchronization are recognized: E_{mutex} and E_{dsync}. E_{mutex} is the set of mutex synchronization edges representing mutual exclusion constraints. Mutex synchronization edges are undirected edges between related Lock and Unlock operations. E_{dsync} is the set of directed synchronization edges representing ordering constraints. These edges join related Set and Wait statements in different threads. Finally, E_{cf} is the set of conflict edges. Conflict edges are directed edges that join any two parallel basic blocks that conflict. Two letter labels on the edge represent the memory operations done at each end of the edge: def (D) or use (U). □

An example of a PFG is shown in Figure 2. Memory access conflicts are represented by dashed edges between the conflicting nodes in each thread (most conflict edges have been removed to improve readability). Mutual exclusion synchronization is represented by dotted edges between Lock and Unlock nodes in concurrent threads.

Definition 2 A path from x to y is a *control path* if it only contains edges in E_{ct}. □

In subsequent sections we will use the standard concepts of dominance and post-dominance [1] applied exclusively to control paths.

3.2. Mutex structures

The concepts and algorithms described in this section are based on the non-concurrency analysis techniques developed by Masticola and Ryder [10]. Our work differs from theirs in the following aspects:

1. Our analysis targets locks instead of binary semaphores.
2. The analysis is intended to gather data flow information for the purposes of program optimization instead of deadlock detection.
3. Even though the notation is similar, there are differences in the definitions and the algorithms used. In particular, we use a simpler notion of mutex body that is not based on the concept of *strict interval* defined by Masticola [9]. Strict intervals require other structural conditions that are not needed in our case. For instance, strict intervals do not include ambiguous or illegal mutex bodies. If at the end of the mutex analysis there is at least one unmatched Lock operation for a lock variable L, the whole set of mutex bodies for L will be discarded. In our case, we allow mutex structures with ill-formed mutex bodies. Our data-flow analysis will still be conservative because illegal mutex bodies in a mutex structure will not be considered when reducing data dependencies.

Definition 3 Given a PFG G, a synchronization variable L and two nodes $n, x \in G$, the set $B_L(n, x) = SDOM^{-1}(n) \bigcap PDOM^{-1}(x)$ is a *mutex body* for L if the following conditions are met:

1. $n =$ Lock(L) and $x =$ Unlock(L),
2. $n\ DOM\ x$ and $x\ PDOM\ n$, and
3. $\forall a \in B_L(n, x)$ such that $a \neq n \wedge a \neq x \Rightarrow a$ is not a Lock(L) or an Unlock(L) node. □

A mutex body defines a single-entry, single-exit region of the graph delimited by nodes n and x. The mutex body includes all the nodes strictly dominated by n and post-dominated by x (i.e., node n is not included in $B_L(n, x)$).

Definition 4 A *mutex structure* for a synchronization object L, denoted M_L, is the set of all mutex bodies $B_L(n, x)$ in the program. □

3.3. Identifying mutex structures in the code

Algorithm A.1 returns the set of all the mutex structures in an explicitly parallel program. The algorithm starts by pairing up Lock and Unlock nodes that comply with conditions 1 and 2 of Definition 3. The final phase of the algorithm (lines 19–26) examines all the mutex bodies found to eliminate those mutex bodies that do not comply with condition 3 of Definition 3. That is, it removes any body found by the previous step that contains `Lock` or `Unlock` nodes for the same variable (other than the entry and exit nodes for the body).

4. CSSA with Mutual Exclusion support

The main goal of mutual exclusion analysis is to reduce the number of incoming conflict edges to nodes in the PFG that use shared variables. In the CSSA framework concurrent modifications to the same memory location by different threads are modeled using π terms which are placed in the *parallel join nodes* of the graph. A parallel join node is one that contains a conflicting use for a shared variable v. Each π term has $n+1$ arguments; one for the unique incoming control flow edge and n for the n incoming conflict edges. The goal of the extensions described in this section is to remove superfluous arguments from π terms inside mutex bodies.

Theorems 1 and 2 give sufficient conditions to reduce the number of reachable definitions for uses inside mutex bodies. Both theorems rely on the concepts of upward-exposed uses [15] and reaching definitions [1].

Theorem 1 Let M_L be a mutex structure for lock variable L. Let D_v^B be a definition for a shared variable v inside a mutex body $B_L(n,x) \in M_L$. If D_v^B does not reach node x then D_v^B cannot reach uses of v in any other mutex body $B'_L(n',x') \in M_L$. □

PROOF Let $U_v^{B'}$ be a use of v in $B'_L(n',x')$. Let d be the node containing D_v^B. Let u be the node containing $U_v^{B'}$. Since d and u are inside mutex bodies in the same mutex structure they cannot execute concurrently. Therefore, the definition in d cannot reach the use in u via the conflict edge that joins d and u. Since $B_L(n,x)$ and $B'_L(n',x')$ cannot execute concurrently, for every execution of the program that includes both mutex bodies there can only be two possible partial orderings between them:

1. $B_L(n,x)$ executes to completion before $B'_L(n',x')$. Even though node d executes before node u, the definition D_v^B cannot reach $U_v^{B'}$ because it is always killed by some other definition before it reaches the exit node of $B_L(n,x)$.
2. $B'_L(n',x')$ executes to completion before $B_L(n,x)$. Node u executes before node d, therefore D_v^B cannot reach $U_v^{B'}$. ■

Theorem 2 Let M_L be a mutex structure for lock variable L. Let U_v^B be a use for a shared variable v inside a mutex body $B_L(n,x) \in M_L$. If U_v^B is not upward-exposed from $B_L(n,x)$ then U_v^B cannot be reached by definitions from any other mutex body $B'_L(n',x') \in M_L$. □

PROOF Let $D_v^{B'}$ be a definition for variable v in mutex body $B'_L(n',x')$. Let d be the node in $B'_L(n',x')$ that contains the definition $D_v^{B'}$. Let u be the node in mutex body $B_L(n,x)$ that contains the use U_v^B. Since d and u are inside mutex bodies in the same mutex structure, they cannot execute concurrently. Therefore, the definition in d cannot reach the use in u via the conflict edge that joins d and u. We need to consider two possibilities:

1. $D_v^{B'}$ does not reach node x'. In this case it is clear that $D_v^{B'}$ cannot reach U_v^B (Theorem 1).
2. $D_v^{B'}$ reaches node x'. Now we need to consider the partial execution ordering between $B_L(n,x)$ and $B'_L(n',x')$:
   (a) $B_L(n,x)$ executes to completion before $B'_L(n',x')$. Node u executes before node d, therefore $D_v^{B'}$ cannot reach U_v^B.
   (b) $B'_L(n',x')$ executes before $B_L(n,x)$. Since U_v^B is not upward-exposed from $B_L(n,x)$, any definitions of v made before $B_L(n,x)$ starts executing are guaranteed to be killed by some other definition inside $B_L(n,x)$. Therefore, $D_v^{B'}$ cannot reach U_v^B. ■

We now introduce the CSSAME form, an extension to the CSSA form to handle mutual exclusion synchronization. Algorithm A.2 transforms an explicitly parallel program P to CSSAME form. The algorithm starts by building the PFG for P. Once the PFG has been built, the algorithm creates the mutex structures for the mutual exclusion synchronization used in the program. The next step builds the CSSA form using the algorithms proposed in [7]. The only difference in our approach is that the underlying sequential SSA form is computed using factored use-def (FUD) chains [15] with appropriate modifications to avoid placing superfluous ϕ terms at `coend` nodes. The computation of partial orderings and the placement of π functions use the same algorithms described in [7]. Notice that since analyzing mutual exclusion synchronization does not require execution ordering information, we do not impose restrictions on the input program.

Once the CSSA form has been computed, π terms are modified using Algorithm A.3. This algorithm examines every mutex body of the program trying to remove arguments from each π term using theorems 1 and 2. A π term will be removed from the graph if and only if at the end of the algorithm it contains only one argument. If the π term only contains one argument, it must be the argument for the incoming control edge to the node because this is the only argument that is never removed by Algorithm A.3.

5. Optimizing explicitly parallel programs

5.1. Constant propagation

Lee *et al.* [7] adapted the sequential Sparse Conditional Constant propagation (SCC) algorithm [14] to work with explicitly parallel programs. We will use the program in Figure 2 to show how our extensions to the original CSSA

```
a0 = 0;
b0 = 0;
cobegin
  T 0: begin
    Lock(L0);
    a1 = 5;
    ta1 = π(a1, a4);
    b1 = ta1 + 3;
    if (b1 > 4) {
      ta11 = π(a1, a4);
      a2 = ta11 + b1;
    }
    a3 = φ(a1, a2);
    ta12 = π(a3, a4);
    x0 = ta12;
    Unlock(L0);
  end

  T 1: begin
    Lock(L0);
    tb0 = π(b0, b1);
    a4 = tb0 + 6;
    ta4 = π(a4, a1, a2);
    y0 = ta4;
    Unlock(L0);
  end
coend
a5 = φ(a3, a4);
print(x0);
print(y0);
```

a. CSSA form

```
a0 = 0;
b0 = 0;
cobegin
  T 0: begin
    Lock(L0);
    a1 = 5;
    b1 = a1 + 3;
    if (b1 > 4) {
      a2 = a1 + b1;
    }
    a3 = φ(a1, a2);
    x0 = a3;
    Unlock(L0);
  end

  T 1: begin
    Lock(L0);
    tb0 = π(b0, b1);
    a4 = tb0 + 6;
    y0 = a4;
    Unlock(L0);
  end
coend
a5 = φ(a3, a4);
print(x0);
print(y0);
```

b. CSSAME form

Figure 3. CSSA forms for the program in Figure 2.

```
a0 = 0;
b0 = 0;
cobegin
  T 0: begin
    Lock(L0);
    a1 = 5;
    ta1 = π(a1, a4);
    b1 = ta1 + 3;
    if (b1 > 4) {
      ta11 = π(a1, a4);
      a2 = ta11 + b1;
    }
    a3 = φ(a1, a2);
    ta12 = π(a3, a4);
    x0 = ta12;
    Unlock(L0);
  end

  T 1: begin
    Lock(L0);
    tb0 = π(b0, b1);
    a4 = tb0 + 6;
    ta4 = π(a4, a1, a2);
    y0 = ta4;
    Unlock(L0);
  end
coend
a5 = φ(a3, a4);
print(x0);
print(y0);
```

a. Using CSSA

```
a0 = 0;
b0 = 0;
cobegin
  T 0:
  begin
    Lock(L0);
    a1 = 5;
    b1 = 8;
    a2 = 13;
    a3 = 13;
    x0 = 13;
    Unlock(L0);
  end

  T 1:
  begin
    Lock(L0);
    tb0 = π(b0, b1);
    a4 = tb0 + 6;
    y0 = a4;
    Unlock(L0);
  end
coend
a5 = φ(a3, a4);
print(x0);
print(y0);
```

b. Using CSSAME

Figure 4. Constant propagation for Figure 2.

framework can be used to improve the constant propagation algorithm when mutual exclusion is taken into account. There are two different CSSA forms for the program in Figure 2. The one in Figure 3a is the original CSSA form without mutual exclusion extensions. Figure 3b shows the CSSAME form built using the algorithms in Section 4 (notice the reduction of π terms in Figure 3b).

Figure 4a shows the result of applying the constant propagation algorithm to the program using CSSA. Notice that the constant propagation is conservatively correct but since the original CSSA framework does not recognize the mutual exclusion semantics of the program, no constants can be propagated. On the other hand, translating the program to CSSAME form allows the compiler to remove all the π terms for variable a in thread T_0. The key to this is the assignment to variable a in thread T_0 right after the lock operation. Since all the statements in thread T_0 execute indivisibly as one atomic operation, uses of variable a after the first assignment cannot possibly be affected by definitions of a made by thread T_1. This allows the compiler to propagate constants inside thread T_0 as if it were a sequential program (Figure 4b).

5.2. Parallel Dead Code Elimination

Dead code refers to program statements that have no effect on any program output [2]. Although it is not common for the programmer to introduce dead code intentionally, dead code may be generated by optimizing transformations [1]. We introduce the Parallel Dead Code Elimination algorithm (PDCE), an extension of the dead code elimination algorithm proposed by Cytron *et al.* [2] to work on explicitly parallel programs. The algorithm starts by marking dead all the statements of the program except those that are assumed to affect the program output such as I/O statements or assignments to variables outside the current scope. This initial set of live statements is used to seed the work list maintained by the algorithm. The list is updated with every new statement that is marked live. When the list empties, all the statements still marked dead are removed from the program. A statement will be marked live if it satisfies one of the following conditions [2]: (1) The statement is assumed to affect the program output. Examples include I/O statements, assignment to global variables, calls to procedures that may have side effects, etc. (2) The statement contains a definition that reaches uses in statements already marked live. (3) The statement is a conditional branch and there are live statements that are control dependent on this conditional branch.

The sequential algorithm needs two important modifications to work on explicitly parallel programs:

1. Condition 2 of Cytron *et al.*'s algorithm calls for the computation of reaching definition information for each live statement of the program. The rationale is that if statement s is live then any other statement that defines variables used by s must also be marked live. We compute reaching definition information using both ϕ and π terms when following use-def chains in the program. Algorithm A.4 computes the

```
b0 = 0;
cobegin
  T 0: begin
    Lock(L0);
    b1 = 8;
    x0 = 13;
    Unlock(L0);
  end

  T 1: begin
    Lock(L0);
    tb0 = π(b0, b1);
    a4 = tb0 + 6;
    y0 = a4;
    Unlock(L0);
  end
coend
print(x0);
print(y0);
```

a. After PDCE

```
b0 = 0;
cobegin
  T 0: begin
    Lock(L0);
    b1 = 8;
    Unlock(L0);
    x0 = 13;
  end

  T 1: begin
    Lock(L0);
    tb0 = π(b0, b1);
    a4 = tb0 + 6;
    Unlock(L0);
    y0 = a4;
  end
coend
print(x0);
print(y0);
```

b. After LICM

Figure 5. PDCE and LICM for Figure 4b.

set of reaching definitions for every use of a variable in an explicitly parallel program. The algorithm is a modified version of an algorithm for finding reaching definitions in a sequential SSA framework [15]. The main modification done to the original algorithm is the additional test for π terms when traversing use-def chains in the PFG. We have also extended the algorithm to compute def-use links (needed by the constant propagation algorithm).

2. A `cobegin` statement will be marked live if there is at least one statement in one of its children threads marked live. If at the end of the algorithm there is only one thread with live statements in it, the `cobegin`/`coend` construct will be replaced by the sequential code corresponding to the live thread.

These modifications to the sequential DCE algorithm are necessary to account for the concurrent activity in the program. Since reaching definition information will be computed using both π and ϕ terms, if a use u is live in one thread, any definition made by other concurrent threads that reach u will also be marked live. Furthermore, the reduction of dependencies made possible by CSSAME directly benefits the elimination of dead code in the program.

To show the effects of dead code elimination consider the program in Figure 2 after constant propagation has been performed (Figure 4b). As can be seen in the example program, all the assignments to variable a in T_0 are dead because they do not affect the output of the program (i.e., they do not reach any other use of a in the program). On the other hand, the assignment to b in T_0 cannot be considered dead because it is used by T_1. Note that a sequential dead code elimination algorithm would have erroneously marked the assignment to b dead because it lacks the appropriate reaching definition information. Figure 5a shows the result of a dead code pass on the code in Figure 4b.

5.3. Lock independent code motion

Because of the restrictions imposed by mutual synchronization operations, it is often desirable to minimize the time spent inside mutex bodies in the program. To achieve this goal we can optimize the code inside mutex bodies as much as possible. Alternatively, we can minimize the amount of code executed inside a mutex body by moving code that does not need to be locked outside the mutex body. In this section we introduce *lock independent code motion* (LICM), a new technique that performs safe code motion on mutex bodies.

To determine what code can be safely moved outside a mutex body we must find those interior statements that are not affected by the presence of the lock. We call these *lock independent* statements. Although it is unlikely for the programmer to write lock independent statements inside a mutex body, other compiler optimizations might produce lock independent code (e.g., the statement $x0 = 13$ in Figure 5a is lock independent due to constant propagation and PDCE). This is similar to the concept of loop-invariant code for standard loop optimization techniques [1]. However, the conditions that make a statement lock independent are different than those that make it loop invariant. Loop invariant computations are basically statements with all their operands constant or with reaching definitions outside the loop. Lock independent code computes the same result whether it is inside a mutex body or not. For instance, a statement that references variables private to the thread will compute the same value whether it is executed inside a mutex body or not. This is also true if the statement references variables not used by any other concurrent thread in the program.

Definition 5 A statement inside a mutex body is *lock independent* if the variables that it defines and/or uses cannot be modified concurrently. □

Although lock independence is a necessary condition to do code motion, it is not sufficient because the motion should also preserve all the control and data dependencies for the statement. For instance, if the statement is inside a loop it cannot be moved out unless the whole loop is lock independent.

To perform code motion we need to modify the flow graph to add two special nodes that will act as landing pads for statements moved out of each mutex body $B_L(n, x)$. We call these two nodes the *pre-mutex* and *post-mutex* node. The *pre-mutex* node is placed as an immediate strict dominator of n, while the *post-mutex* node is placed as an immediate strict post-dominator of x.

Theorem 3 Let s be a lock independent statement inside a mutex body $B_L(n, x)$. Let a be the node containing s:

1. If a dominates all the nodes in B and s does not have any reaching definitions within a then s can be moved to the

pre-mutex node of B.

2. If x immediately post-dominates a and s does not have any reached uses within a then s can be moved to the post-mutex node of B. □

PROOF 1. If a dominates all the nodes in B then its immediate dominator must be node n. If s does not have any reaching definitions within a then s can be moved to a node dominating a without affecting its internal data dependencies. Furthermore, definitions reaching s cannot reach through conflict edges because s is lock independent. Neither can they reach from node n because there are no definitions in that node. Therefore, moving s to the pre-mutex node will not alter any data dependencies in the program. Notice that when moving s to the pre-mutex node, it should be placed as the last statement of the node. This will preserve any data dependencies from statements already present in the node.

2. If x immediately post-dominates a then a is the last node to be executed before leaving the mutex body. If definitions made by s do not reach any use within a then moving s to the post-mutex node will not alter any data dependencies inside a. Furthermore, definitions make by s cannot reach other threads through conflict edges because s is lock independent. Therefore, moving s to the post-mutex node will not alter any data dependencies in the program. Notice that when moving s to the post-mutex node, it should be placed as the first statement of the node. This will preserve any data dependencies to statements already present in the node. ■

These conditions guarantee that the statement being moved will not break any data dependencies with other nodes in the body and will not introduce any conflict edges with any concurrent node. Applying Theorem 3 to the program in Figure 5a allows the compiler to move some statements out of the mutex bodies to obtain the equivalent program in Figure 5b. Notice that both assignments to variables x and y can be safely moved out of each mutex body because there are no conflicting definitions in their sibling threads. Algorithm A.5 implements the concepts described previously. After code motion is complete, any empty mutex bodies will be removed from the program.

6. Implementation

The algorithms discussed in previous sections have been implemented in a prototype compiler for the C language using the SUIF compiler system [4]. To avoid modifying SUIF's front-end we added support for `cobegin`/`coend` and `doall` parallel structures via language macros. These macros re-define control structures of the language so that the compiler can recognize them as parallel at the intermediate language level.

Once the program has been parsed by the SUIF front-end, the compiler creates the corresponding PFG and its CSSAME form. The PFG implementation is an extension of the sequential Control Flow Graph library provided by Machine SUIF [5]. The PFG can be displayed using a variety of graph visualization systems. The flow graphs in this paper were generated with the VCG tool (Visualization of Compiler Graphs) [12]. The CSSA form for the program can also be displayed as an option. Mutual exclusion analysis can also issue warning messages like unmatched `Lock` and `Unlock` operations or improperly nested locks. A limited form of data race detection capability is also built-in for inconsistent use of locks to protect shared variables. For instance, if modifications to a variable are not always protected by the same lock, the compiler will warn the user about a potential data race.

A simple extension to algorithm A.1 allows the compiler to perform some semantic checking on the synchronization structure of the program. At the end of the algorithm, every `Lock` or `Unlock` node in $p_i^{lock} \bigcup p_i^{unlock}$ that is not part of a mutex body can be reported as a warning to the user. The compiler will recognize several potentially unsafe situations and report a warning.

7. Future work

We have found that the CSSAME form facilitates the translation of scalar optimizations to the parallel case, especially if the sequential strategy is SSA based. We are presently investigating the representation of parallel loops in the CSSA framework. Different semantics for parallel loops (i.e., `doaccross`, `doall`, etc.) will have different data-flow properties. Another extension the CSSAME framework involves other commonly used synchronization primitives such as barriers and semaphores.

With the lock independent code motion strategy we have entered the field of new optimization techniques that are specifically targeted at explicitly parallel programs. We are presently designing new optimization techniques that take advantage of the parallel and synchronization structure of these programs.

A. Algorithms

A.1. Identification of mutex structures

Input: A PFG G and a set $L = \{L_1, L_2, \ldots, L_m\}$ containing all the lock variables used in the program.

Output: A set of mutex structures $M = \bigcup_i M_i$ where M_i is the set of mutex bodies for lock variable L_i.

```
/* Find nodes in $G$ that lock and unlock each $L_i$ */
foreach lock variable $L_i$ do
    $p_i^{lock} \leftarrow \{n \in N : n = Lock(L_i)\}$
    $p_i^{unlock} \leftarrow \{x \in N : x = Unlock(L_i)\}$
end for

/* Build the dominator and post-dominator trees for $G$ */
call buildDomTree($G$)
call buildPDomTree($G$)
/* Find candidate mutex bodies */
foreach lock variable $L_i$ do
    foreach $n \in p_i^{lock}$ do
        foreach $x \in p_i^{unlock}$ do
            if $n \in DOM(x)$ and $x \in PDOM(n)$ then
                add $(n, x)$ to the set of candidates $M_i$
            end if
        end for
    end for
end for
/* Remove illegal mutex bodies from each $M_i$ */
foreach $(n, x) \in M_i$ do
    foreach $m \in p_i^{lock} \bigcup p_i^{unlock}$ do
        if $m \neq n$ and $m \neq x$ and $n \in DOM(m)$ and $x \in PDOM(m)$ then
            remove $(n, x)$ from $M_i$
        end if
    end for
end for
$M \leftarrow \bigcup_i M_i$
return $M$
```

A.2. CSSAME algorithm

Input: An explicitly parallel program P
Output: The program P in CSSAME form

```
Build the PFG for $P$ using an extended version of the CFG algorithm in [5]
Identify mutex structures using Algorithm A.1.
Compute the CSSA form for the graph using the algorithms in [7].
Rewrite $\pi$ terms using Algorithm A.3.
```

A.3. Rewrite π terms

Input: A PFG G in CSSA form
Output: The graph G in CSSA form with π terms modified to account for mutual exclusion synchronization

```
/* Traverse mutex bodies looking for $\pi$ terms to rewrite */
foreach lock variable $L_i$ do
    foreach mutex body $b \in MutexStruct(L_i)$ do
        call $rewrite(b)$
    end for
end for

/* Examine all the $\pi$ terms in $b$ */
procedure $rewrite(b)$
foreach node $n \in b$ do
    foreach $\pi$ term $p \in n$ do
        $v$ is the variable referenced by $p$
        /* If an argument complies with theorems 1 or 2, */
        /* then we may safely remove the argument from the $\pi$ term */
        foreach $p$ argument $d$ coming from a conflict edge do
            if $d$ comes from another mutex body $b' \in MutexStruct(b)$ then
                if (the use of $v$ is not upward exposed from $b$) or ($d$ does not reach the exit node of $b'$) then
                    Remove $d$ from $p$
                end if
            end if
        end for
        /* If $\pi$ term $p$ has no conflict arguments then remove it */
        if $p$ has only one argument then
            $chain(u) \leftarrow$ first argument of $p$
            Remove $p$ from $n$
        end if
    end for
end for
```

A.4. Parallel reaching definitions

Input: A PFG G in CSSAME form
Output: The set of reaching definitions for each variable used in the program and the set of reached uses for each variable defined in the program

```
foreach variable definition $d$ in the program do
    $marked(d) \leftarrow \perp$
    $uses(d) \leftarrow \emptyset$
end for
foreach variable use $u$ in the program do
    $defs(u) \leftarrow \emptyset$
    call followChain(chain(u), u)
end for

procedure $followChain(d, u)$
if $marked(d) = u$ then
    return
end if
$marked(d) \leftarrow u$
if $d$ is a definition for $u$ then
    Add $d$ to $defs(u)$
    Add $u$ to $uses(d)$
end if
if ($d$ is a $\phi$ term) or ($d$ is a $\pi$ term) then
    foreach term argument $j$ do
        call followChain($j, u$)
    end for
end if
```

A.5. Lock independent code motion

Input: A PFG G in CSSAME form
Output: The graph with lock independent code moved to the corresponding pre-mutex and post-mutex nodes

```
foreach lock variable $L_i$ do
    foreach mutex body $B_{L_i}(n, x) \in MutexStruct(L_i)$ do
        insert pre-mutex node immediately dominating $n$
        insert post-mutex node immediately post-dominating $x$
    end for
end for

foreach lock variable $L_i$ do
    foreach mutex body $B_{L_i}(n, x) \in MutexStruct(L_i)$ do
        $PRE \leftarrow$ the pre-mutex node of $B$
        $POST \leftarrow$ the post-mutex node of $B$

        done $\leftarrow$ FALSE
        while not done do
            $a \leftarrow$ node immediately dominated by $n$
            foreach statement $s \in a$ do
                if $s$ is lock independent then
                    if $Definers(s)$ does not contain a statement from $a$ then
                        move $s$ to the end of node $PRE$
                    end if
                end if
            end for
            if $a = \emptyset$ then
                remove $a$ from the graph
            else
                $done \leftarrow$ TRUE
            end if
        end while

        done $\leftarrow$ FALSE
        while not done do
```

```
      $b \leftarrow$ node immediately post-dominated by $x$
      **foreach** statement $s \in b$ **do**
        **if** $s$ is lock independent **then**
          **if** $Users(s)$ does not contain a statement from $b$ **then**
            move $s$ to the beginning of node $POST$
          **end if**
        **end if**
      **end for**
      **if** $b = \emptyset$ **then**
        remove $b$ from the graph
      **else**
        $done \leftarrow$ TRUE
      **end if**
    **end while**

    **if** $DOM^{-1}(n) \bigcap PDOM^{-1}(x) = \emptyset$ **then**
      remove $n$ and $x$ from the graph
    **end if**
  **end for**
**end for**
```

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading, Mass.: Addison-Wesley, Reading, MA, second edition, 1986.

[2] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, Oct. 1991.

[3] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. *ACM SIGPLAN Notices*, 28(7):159–168, July 1993.

[4] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[5] G. Holloway and C. Young. The flow analysis and transformation libraries of Machine SUIF. In *Proc. 2nd SUIF Compiler Workshop*, Stanford University, Aug. 1997.

[6] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *TOPLAS*, 18(3):268–299, May 1996.

[7] J. Lee, S. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proc 10th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1997.

[8] D. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1):25–43, 1993.

[9] S. Masticola. *Static Detection of Deadlocks in Polynomial Time*. PhD thesis, Department of Computer Science, Rutgers University, 1993.

[10] S. Masticola and B. Ryder. Non-concurrency analysis. In *Proc 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[11] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Int'l Conference on Parallel Processing*, volume II, pages 105–113, Aug. 1990.

[12] G. Sander. Graph layout through the VCG tool. In R. Tamassia and I. G. Tollis, editors, *Proc. Graph Drawing, DIMACS International Workshop GD'94, Lecture Notes in Computer Science 894*, pages 194–205. Berlin: Springer Verlag, 1995.

[13] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *TOPLAS*, 10(2):282–312, Apr. 1988.

[14] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *TOPLAS*, 13(2):181–210, Apr. 1991.

[15] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Reading, Mass.: Addison-Wesley, 1996.